hw3 - 20181755 이건희

과제에서 요구한 내용은 가장 마지막 페이지에 나옵니다.

본 보고서는 제 스스로의 공부를 위해서 얽혀있는 생각들을 정리하고 이를 글로써 기록한 거라, 읽어보실만한 가치가 크게 있는지는 잘 모르겠습니다.

어떤 올바른 문장은 파싱 결과로 올바른 트리를 구성할 것이다. 트리 자료구조는 레벨이 존재하는데, 설명의 편의를 위해 이 레벨을 사용하여 declaration을 level 0으로 두고, 루트 노드로부터의 최단 거리를 level로 두겠다.

서술 규칙 1.
각 요소의 레벨을 나타내기 위해
①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰⑱⑲⑳
과 같이 원형 기호를 사용하겠다.

서술 규칙 2.
어떤 요소의 자식 노드가 없을 경우, 해당 요소를
② auto
와 같이 볼드체로 나타내겠다.

lv 0.
0-1. ⓪ declaration은
① declaration_specifiers ;
① declaration_specifiers ① init_declarator_list ;
의 두 형태로 존재한다.

과제는 declaration_specifiers, declarator에 대해서 설명하라 했는데, declarator는 declaration을 직접적으로 구성하지는 않는다. declaration이 init_declarator가 되고, init_declarator(들)이 모여 init_declarator_list이 되기 때문이다.

따라서 declarator가 몇 레벨인지 우선적으로 알아보겠다.

들어가기 전에 : ① init_declarator_list

---

lv 1.
1-1. ① init_declarator_list는 (0-1)
② init_declarator
① init_declarator_list ① , ② init_declarator
의 두 형태로 존재한다. 이 중 두 번째는 ① init_declarator_list를 정의하는데 그 자신이 나왔다. 이는 ① init_declarator_list는 얼마든지 반복이 가능하여, "a, b = 20, c, d"와 같이 나와도 문법적으로 맞다는 것이다.

lv 2.
2-1. ② init_declarator는 (1-1)
③ declarator (초기화 X)
③ declarator ② = ③ initializer (초기화 O)
의 두 형태로 존재한다.

lv 3.
3-1. ③ declarator - 과제로 설명해야 할 것 -> 분석 2에서 설명

3-2. ③ initializer는 (2-1-2)
④ constant_expression
③ { ④ initializer_list **}**
의 두 형태로 존재한다. 첫 번째의 ④ constant_expression은 분석 2에서 설명하겠다.

lv 4.
4-1. ④ initializer_list는 (3-2-2)
③ initializer
④ initializer_list ④ , ③ initializer
의 두 형태로 존재한다. 두 번째는 ④ initializer_list를 정의하는데 그 자신이 나왔다. 이는 ④ , ③ initializer가 얼마든지 반복이 가능하여, "int arr[3] = {1, 2, 3}"과 같이 한 번에 여러 변수의 초기화가 가능하다는 것이다.

분석 1 : ① declaration_specifiers

lv 1.
1-1. ① declaration_specifiers는
② type_specifier 단 하나 (“int”)
② type_specifier ① declaration_specifiers
(“int float” - 재귀적; 얼마든지 반복 가능)
② storage_class_specifier 단 하나 (“static”)
② storage_class_specifier와 ① declaration_specifiers
(“auto static” - 재귀적이라 얼마든지 반복 가능)
의 네 형태로 존재한다. 이중 두 번째와 네 번째는 ① declaration_specifiers를 정의하는데 그 자신이 나왔다. 이는 ② type_specifier와 ② storage_class_specifier가 얼마든지 반복이 가능하여. “int auto char static”과 같이 나와도 문법적으로 맞다는 것이다(의미적으로는 틀림).

lv 2.
2-1. ② type_specifier는 (1-1-1~2)
③ struct_specifier
③ enum_specifier
③ TYPE_IDENTIFIER
의 세 형태로 존재한다.

2-2. ② storage_class_specifier은 (1-1-3~4)
② auto
② static
② typedef
의 세 형태로 존재한다.

lv 3.
3-1. ③ struct_specifier는 (2-1-1)
④ struct_or_union **③ IDENTIFIER**
④ struct_or_union **③ {** ④ struct_declaration_list **}**
④ struct_or_union **③ IDENTIFIER ③ {** ④ struct_declaration_list **}**
의 세 형태로 존재한다.

3-2. ③ enum_specifier는 (2-1-2)
③ enum ③ IDENTIFIER
③ enum ③ { ④ enumerator_list **}**
③ enum ③ IDENTIFIER ③ { ④ enumerator_list **}**
의 세 형태로 존재한다.

3-3. ③ TYPE_IDENTIFIER는 (2-1-3) 편의상
③ void
③ char
③ int
③ float
의 네 형태로만 존재하기로 교재에서 약속하였다. 이미 알고있는 자료형이라 생각하면 된다.

3-1, 3-2에서 **③ IDENTIFIER**는 고유한 이름이다.

lv 4.
4-1. ④ struct_or_union은 (3-1)
④ struct
④ union
의 두 형태로 존재한다. 당연하게도 이는 구조체, 공용체를 뜻한다.

4-2. ④ struct_declaration_list은 (3-1-2~3)
⑤ struct_declaration
④ struct_declaration_list ⑤ struct_declaration
의 두 형태로 존재한다.

4-3. ④ enumerator_list은 (3-2-2~3)
⑤ enumerator
④ enumerator_list, ⑤ enumerator
의 두 형태로 존재한다.

4-2, 4-3의 두 번째는 각각을 정의하는데 그 자신이 나왔다. 이는 ⑤ struct_declaration, ⑤ enumerator가 얼마든지 반복이 가능하여, “struct s = {float f, char c};”, “enum e = {ONE = 1, TWO, THREE};”와 같이 한 구조체나 열거형을 여러 멤버로 구성이 가능하다는 것이다.

lv 5.
5-1. ⑤ struct_declaration은 (4-2-2) (”float f“)
② type_specifier ⑥ struct_declarator_list (declaration이 아님에 유의)
의 한 형태로 존재한다.

5-2. ⑤ enumerator는 (4-3-2) (“ONE = 1”)
⑤ IDENTIFIER
⑤ IDENTIFIER ⑤ = ⑥ constant_expression
의 두 형태로 존재한다. **⑤ IDENTIFIER**는 고유한 이름이다. ⑥ constant_expression은 분석 2에서 설명하겠다.

lv 6.
6-1. ⑥ struct_declarator_list는 (5-1)
⑦ struct_declarator
⑥ struct_declarator_list **⑥ ,** ⑦ struct_declarator
의 두 형태로 존재한다. 두 번째는 ⑥ struct_declarator_list를 정의하는데 그 자신이 나왔다. 이는 ⑦ struct_declarator가 얼마든지 반복이 가능하다는 것이다.

lv 7.
7-1. ⑦ struct_declarator은 (6-1)
③ declarator
의 한 형태로 존재한다. ⑦ struct_declarator를 정의하는데 상위 레벨의 ③ declarator가 나왔다.

이를 이해하려면 ③ declarator에 대한 분석이 필요하다.

분석 2 : ③ declarator

---

lv 3.
3-1. ③ declarator은
④ direct_declarator
④ pointer ④ direct_declarator
의 두 형태로 존재한다.

lv 4.
4-1. ④ direct_declarator는 (3-1)
④ IDENTIFIER
④ (③ declarator **)**
④ direct_declarator **④ [** ⑤ constant_expression_opt **]**
④ direct_declarator **④ (** ⑤ parameter_type_list_opt **)**
의 네 형태로 존재하는데, 이들을 제대로 이해할 필요가 있다.

첫 번째의 **④ IDENTIFIER**는 고유한 이름(변수명)이다.
두 번째는 ④ direct_declarator을 정의하는데 상위 레벨의 ③ declarator이 나왔는데, 여기서 소괄호는 두 가지의 용도가 있다.
1. "(i)" : 단순히 변수명에 괄호를 씌움
2. "(*fun)(int, float)" - parenthesized declarator (정확한 용어는 구글링으로 찾음)

```
/** "int (*fun)(int, float);"와 "int *fun(int, float);"는 다르다 */
"int (*fun)(int, float);" - 매개변수가 int, float이고, int형을 return하는 함수에 대한 포인터
"int *fun(int, float);" - 매개변수가 int, float이고, int*형을 return하는 함수
```

세 번째는 대괄호가 나온 것으로 보아 배열이라는 것을 알 수 있다. 또한 ④ direct_declarator를 정의하는데 그 자신이 나왔다. 이는 [⑤ constant_expression_opt]가 얼마든지 반복이 가능하여, "arr[][][]"와 같이 나와도 문법적으로는 맞다는 것이며, 실제로 이는 다차원 배열을 의미한다.
네 번째는 소괄호가 나온 것으로 보아 함수라는 것을 알 수 있다. 또한 ④ direct_declarator를 정의하는데 그 자신이 나왔다. 이는 (⑤ parameter_type_list_opt)가 얼마든지 반복이 가능하여, "fun()()"와 같이 나와도 문법적으로는 맞다는 것이다(의미적으로는 틀림).
세 번째와 네 번째의 재귀적인 ④ direct_declarator의 용도는 의미적으로 따져보았을 때 일반적으로 그 자리에 **④ IDENTIFIER**로 배열이나 함수의 이름을 붙이기 위함이라고 생각하는 것이 타당할 것이다.

4-2. ④ pointer는 (3-1-2)
④ *
④ * ④ pointer
의 두 형태로 존재한다. 두 번째는 ④ pointer를 정의하는데 그 자신이 나왔다. 이는 ④ *가 얼마든지 반복이 가능하여, "**x", "***y"와 같이 나와도 문법적으로 맞다는 것이며, 이는 다중 포인터를 의미한다.

lv 5.
5-1. ⑤ constant_expression_opt는 (4-2-3)
⑤ λ
⑥ constant_expression
의 두 형태로 존재한다.

5-2. ⑤ parameter_type_list_opt는 (4-2-4)
⑤ λ
⑥ parameter_type_list
의 두 형태로 존재한다.

각각은 **⑤ λ**의 형태로도 존재할 수 있는데, 이름처럼 optional하기 때문이다. 배열의 크기를 지정하지 않는다면 가변길이배열(C99표준)이 될 것이고, "fun()"와 같이 매개변수가 없는 함수를 인자 없이 호출할 수 있기 때문이다.

여기까지 진행한 결과, level 6으로는 ⑥ constant_expression (5-1-2), ⑥ parameter_type_list (5-2-2)가 존재한다. 각 서브트리가 의미적으로 너무나 다르기에 각 케이스를 나눠서 분석하도록 하겠다.

```
case 1 : ⑥ constant_expression (5-1-2)
case 2 : ⑥ parameter_type_list (5-2-2)
```

case 1, lv 6.
6-1. ⑥ constant_expression는 (5-1-2)
⑦ expression
의 한 형태로 존재한다.

case 1, lv 7.
7-1. ⑦ expression은 (6-1)
⑧ assignment_expression
의 한 형태로 존재한다.

case 1, lv 8.
8-1. ⑧ assignment_expression은 (7-1)
⑨ logical_or_expression
⑨ unary_expression **⑧ =** ⑧ assignment_expression
의 두 형태로 존재한다. 두 번째는 ⑧ assignment_expression를 정의하는데 그 자신이 나왔다. 이는 ⑨ unary_expression **⑧ =**이 얼마든지 반복이 가능하여, "1=2=3"과 같이 나와도 문법적으로는 맞다는 것이다.

이것이 의미적으로 맞는지 점검해보기 위해 다음과 같은 간단한 코드를 만들었다.

```
...
    int a = 0, b = 0, c = 1;

    a=b=c;

    printf("%d %d %d", a, b, c);
...
```

이 코드는 정상작동하는 코드이다. 따라서, 위의 예시는 의미적으로도 맞다.
연산자의 결합 순서가 우측-좌측이기에, b=c, a=b(c의 값이 저장됨) 순으로 연산이 이루어진다.

여기까지 진행한 결과, level 9로는 ⑨ logical_or_expression (8-1-1), ⑨ unary_expression (8-1-2)가 존재한다. 각 서브트리가 의미적으로 너무나 다르기에 각 케이스를 나눠서 분석하도록 하겠다.

```
case 1-1 : ⑥ constant_expression (5-1-2) & ⑨ logical_or_expression (8-1-1)
case 1-2 : ⑥ constant_expression (5-1-2) & ⑨ unary_expression (8-1-2)
```

case 1-1, lv 9.
9-1. ⑨ logical_or_expression는 (8-1-1)
⑩ logical_and_expression
⑨ logical_or_expression **⑨ ||** ⑩ logical_and_expression
의 두 형태로 존재한다. 두 번째는 ⑨ logical_or_expression를 정의하는데 그 자신이 나왔다. 이는 **⑨ ||** ⑩ logical_and_expression이 얼마든지 반복이 가능하여, "(a || b || c || d)"와 같이 나와도 문법적으로 맞다. 의미적으로도 "a이거나 b이거나 c이거나 d"라는 뜻으로 맞는다.

case 1-1, lv 10.
10-1. ⑩ logical_and_expression은 (9-1)
⑪ equality_expression
⑩ logical_and_expression **⑩ &&** ⑪ equality_expression
의 두 형태로 존재한다. 두 번째는 ⑩ logical_and_expression을 정의하는데 그 자신이 나왔다. 이는 **⑩ &&** ⑪ equality_expression이 얼마든지 반복이 가능하여, "(a && b && c && d)"와 같이 나와도 문법적으로 맞다. 의미적으로도 "a이면서 b이면서 c이면서 d"라는 뜻으로 맞는다.

case 1-1, lv 11.
11-1. ⑪ equality_expression은 (10-1)
⑫ relational_expression
⑪ equality_expression **⑪ ==** ⑫ relational_expression
⑪ equality_expression **⑪ !=** ⑫ relational_expression
의 세 형태로 존재한다. 두 번째와 세 번째는 ⑪ equality_expression을 정의하는데 그 자신이 나왔다. 이는 (**⑪ ==** or **⑪ !=**) ⑫ relational_expression이 얼마든지 반복이 가능하여, "a==b!=c"와 같이 나와도 문법적으로 맞다.

이것이 의미적으로 맞는지 점검해보기 위해 다음과 같은 간단한 코드를 만들었다.

```
...
    int a = 1, b = 2, c = 1;

    if (a!=b==c){
        printf("nice");
    }
...
```

이 코드는 정상작동하는 코드이다. 따라서, 위의 예시는 의미적으로도 맞다.
연산자의 결합 순서가 좌측-우측이기에, a!=b, (a!=b)=c 순으로 비교가 이루어진다.
관계연산자가 들어간 표현은 일반적으로 int로 취급한다.

case 1-1. lv 12.
12-1. ⑫ relational_expression은 (11-1)
⑬ additive_expression
⑫ relational_expression ⑫ < ⑬ additive_expression
⑫ relational_expression ⑫ > ⑬ additive_expression
⑫ relational_expression ⑫ <= ⑬ additive_expression
⑫ relational_expression ⑫ >= ⑬ additive_expression
의 다섯 형태로 존재한다. 두 번째부터 다섯 번째는 ⑫ relational_expression을 정의하는데 그 자신이 나왔다. 이는 (⑫ < or ⑫ > or ⑫ <= or ⑫ >=) ⑬ additive_expression이 얼마든지 반복이 가능하여 "1 >= 2 < 3"와 같이 나와도 문법적으로 맞다.

이것이 의미적으로 맞는지 점검해보기 위해 다음과 같은 간단한 코드를 만들었다.
```
…
    int a = 1, b = 2, c = 1;

    if (a<b<=c){
        printf("nice"); // printed
    }

    if (a<b<c){
        printf("nice"); // not printed
    }
…
```
이 코드는 정상작동하는 코드이다. 따라서, 위의 예시는 의미적으로도 맞다.
연산자의 결합 순서가 좌측-우측이기에, a<b, (a<b)<=c의 순으로 비교가 이루어진다.
관계연산자가 들어간 표현은 일반적으로 int로 취급한다.

case 1-1, lv 13.
13-1. ⑬ additive_expression은 (12-1)
⑭ multiplicative_expression
⑬ additive_expression ⑬ + ⑭ multiplicative_expression
⑬ additive_expression ⑬ - ⑭ multiplicative_expression
의 세 형태로 존재한다. 두 번째와 세 번째는 ⑬ additive_expression을 정의하는데 그 자신이 나왔다. 이는 (⑬ + or ⑬ -) ⑭ multiplicative_expression이 얼마든지 반복이 가능하여 "1+2-3"와 같이 나와도 문법적으로 맞고, 의미적으로도 상식적으로 맞다.

case 1-1, lv 14.
14-1. ⑭ multiplicative_expression은 (13-1)
⑮ cast_expression
⑭ multiplicative_expression ⑭ * ⑮ cast_expression
⑭ multiplicative_expression ⑭ / ⑮ cast_expression
⑭ multiplicative_expression ⑭ % ⑮ cast_expression
의 네 형태로 존재한다. 두 번째부터 네 번째는 ⑭ multiplicative_expression 정의하는데 그 자신이 나왔다. 이는 (⑭ * or ⑭ / or ⑭ %) ⑮ cast_expression이 얼마든지 반복이 가능하여 "10*2%3"와 같이 나와도 문법적으로 맞고, 의미적으로도 상식적으로 맞다.

case 1-1, lv 15.
15-1. ⑮ cast_expression (14-1)
바로 다음 단락 case 1-2에 자세한 분석이 나온다.

case 1-1 : ⑥ constant_expression (5-1-2) & ⑨ logical_or_expression (8-1-1)
case 1-2 : ⑥ constant_expression (5-1-2) & ⑨ unary_expression (8-1-2)

case 1-2, lv 9.
9-1. ⑨ unary_expression는 (8-1-2)
⑨ ++ ⑨ unary_expression
⑨ -- ⑨ unary_expression
⑨ sizeof ⑨ unary_expression
⑨ & ⑩ cast_expression
⑨ * ⑩ cast_expression
⑨ ! ⑩ cast_expression
⑨ + ⑩ cast_expression
⑨ - ⑩ cast_expression
⑨ sizeof (⑩ type_name)
⑩ postfix_expression
의 열 형태로 존재한다. 첫 번째부터 세 번째는 ⑨ unary_expression을 정의하는데 그 자신이 나왔지만, "++++a"와 같이 주어졌다면 문법적으로는 맞으나 의미적으로는 틀리다. 상식적으로 ⑨ sizeof (⑩ type_name)와 함께 "sizeof (a++)"식으로 사용해야 의미적으로 맞을 것이다.

9-2. ⑨ cast_expression는 (9-1-4~8)
⑨ unary_expression
⑨ (⑩ type_name) ⑨ cast_expression
의 두 형태로 존재한다. 두 번째는 ⑨ cast_expression을 정의하는데 그 자신이 나왔다. 이는 (⑩ type_name)이 얼마든지 반복이 가능하여 "(int) (short) (char) a"와 같이 나와도 문법적으로 맞다.

이것이 의미적으로 맞는지 점검해보기 위해 다음과 같은 간단한 코드를 만들었다.
```
…
    char c1 = 'a';

    char c2 = (char) (short) (int) c1;

    printf("%c %c", c1, c2);
…
```
이 코드는 정상작동하는 코드이다. 따라서, 위의 예시는 의미적으로도 맞다.
연산자의 결합 순서가 우측-좌측이기에, int, short, char 순으로 캐스팅이 이루어진다.

case 1-2, lv 10.
10-1. ⑩ type_name은 (9-1-9, 9-2-2)
① declaration_specifiers
① declaration_specifiers ⑪ abstract_declarator
의 두 형태로 존재한다.

10-2. ⑩ postfix_expression은 (9-1-10)
⑪ primary_expression
⑩ postfix_expression ⑩ [⑦ expression]
⑩ postfix_expression ⑩ (⑪ arg_expression_list_opt)
⑩ postfix_expression ⑩ . ⑩ IDENTIFIER
⑩ postfix_expression ⑩ -> ⑩ IDENTIFIER
⑩ postfix_expression ⑩ ++
⑩ postfix_expression ⑩ --
의 일곱 형태로 존재한다. 이중 두 번째부터 일곱 번째 까지는 ⑩ postfix_expression을 정의하는데 그 자신이 나왔다. 이는 각각의 자신 이외의 부분을 얼마든지 반복하여도 문법적으로는 틀리지 않으나, 의미적으로 "a++++"와 같은 것은 불가능하다. 상식적인 선에서 a->x++와 같은 조합을 생각해 볼 수 있을 것이다. 또한 두 번째에서 ⑩ postfix_expression을 정의하는데 상위 레벨의 ⑦ expression이 나왔다. 네 번째와 다섯 번째의 ⑩ IDENTIFIER은 어떤 구조체/공용체의 고유한 맴버명이다.

case 1-2, lv 11
11-1. ⑪ primary_expression은 (10-2-1)
⑪ IDENTIFIER
⑪ INTEGER_CONSTANT
⑪ FLOAT_CONSTANT
⑪ STRING_LITERAL
⑪ (⑦ expression)
의 다섯 형태로 존재한다. 다섯 번째는 ⑪ primary_expression을 정의하는데 상위 레벨의 ⑦ expression이 나왔다. 또한 첫 번째부터 네 번째까지는 상수이다.

11-2. ⑪ arg_expression_list_opt은 (10-2-3)
⑪ λ
⑫ arg_expression_list
의 두 형태로 존재한다. ⑪ λ의 형태로도 존재할 수 있는데, 이름처럼 optional하기 때문이다. 함수는 인자를 받지 않게끔 "fun(void)"와 같이 매개변수 없이 정의될 수 있다.

case 1-2, lv 12.
12-1. ⑫ arg_expression_list는 (11-2-2)
⑧ assignment_expression
⑫ arg_expression_list ⑫ , ⑧ assignment_expression
의 두 형태로 존재한다. 두 번째는 ⑫ arg_expression_list를 정의하는데 그 자신이 나왔다. 이는 ⑫ , ⑧ assignment_expression이 얼마든지 반복이 가능하여 "fun (a, 'a', "a")"와 같이 함수에 여러 인자의 전달이 가능하다. 두 형태 모두 ⑫ arg_expression_list을 정의하는데 상위 레벨의 ⑧ assignment_expression이 사용되었다.

case 1-2를 이해하는데 있어서 가장 중요한 내용이다.

⑧ assignment_expression은 위에서 분석했기로
⑨ logical_or_expression
⑨ unary_expression ⑧ = ⑧ assignment_expression
의 두 형태로 존재하는데
첫 번째의 ⑨ logical_or_expression에서 자식 노드를 향해 타고 내려가다보면
결국 ⑨ unary_expression이 나온다.

이 ⑨ unary_expression이란 무엇인가? 사전적인 정의는 "단항 표현"이다.

⑨ unary_expression에서 자식 노드를 향해 타고 내려가면서 만나는 매 노드마다 적당한 예시를 만들어보면, "단항 표현"이라는 사전적인 정의에 동의를 할 수밖에 없을 것이다. 이와 동시에, ⑥ constant_expression의 "상수 표현"이라는 사전적인 정의에 동의를 할 수밖에 없을 것이다.

5-1. ⑤ constant_expression_opt는 (4-2-3)
⑤ λ
⑥ constant_expression

5-2. ⑤ parameter_type_list_opt는 (4-2-4)
⑤ λ
⑥ parameter_type_list

case 1 : ⑥ constant_expression (5-1-2)
case 2 : ⑥ parameter_type_list (5-2-2)

case 2, lv 6.
6-1. ⑥ parameter_type_list는 (5-2-2)
⑦ parameter_list
⑦ parameter_list ⑥ , ...
의 두 형태로 존재한다. 두 번째는 함수가 가변적인 개수의 인자를 받는다는 뜻이다.

case 2, lv 7.
7-1. ⑦ parameter_list는 (6-1)
⑧ parameter_declaration
⑦ parameter_list ⑦ , ⑧ parameter_declaration
의 두 형태로 존재한다. 두 번째는 ⑦ parameter_list을 정의하는데 그 자신이 나왔다. 이는 ⑦ , ⑧ parameter_declaration이 얼마든지 반복이 가능하여 (int, char)와 같이 나와도 문법적으로 맞다. 상식적으로 함수를 정의할 때 매개변수를 얼만큼 설정할지는 프로그래머의 몫이고, 개수의 제한 역시나 없기에 의미적으로도 맞다.

case 2, lv 8.
8-1. ⑧ parameter_declaration은 (7-1)
① declaration_specifiers
① declaration_specifiers ③ declarator
① declaration_specifiers ⑨ abstract_declarator
의 세 형태로 존재한다.

case 2, lv 9.
9-1. ⑨ abstract_declarator는 (8-1)
④ pointer
⑩ direct_abstract_declarator
④ pointer ⑩ direct_abstract_declarator
의 세 형태로 존재한다.

case 2, lv 10. (9-1)
10-1. ⑩ direct_abstract_declarator는
⑩ (⑨ abstract_declarator **)**
⑩ [⑤ constant_expression_opt **]**
⑩ (⑤ parameter_type_list_opt **)**
⑩ direct_abstract_declarator **⑩ [** ⑤ constant_expression_opt **]**
⑩ direct_abstract_declarator **⑩ (** ⑤ parameter_type_list_opt **)**
의 다섯 형태로 존재한다. 첫 번째는 ⑩ direct_abstract_declarator을 정의하는데 상위 레벨의 ⑨ abstract_declarator가 나왔는데, 여기서 소괄호의 용도는 분석 2의 lv 4를 참고 바란다. 두 번째부터 다섯 번째의 ⑤ constant_expression_opt, ⑤ parameter_type_list_opt에 대한 설명은 분석 2의 case 1, case 2 각각 lv 6을 참고 바라며, 이들의 재귀성에 대한 설명은 분석 2의 lv 4를 참고 바란다.

이렇게 모든 레벨의 노드를 전부 분석한 이유는, 구성 요소들의 정의가 재귀적이거나 사이클을 이루기 때문이다. 전체를 한번 다 둘러보는 분석이 없었다면 설명은 불가능했을 것이다.

설명 1. declaration_specifiers

딱 잘라서 한 문장으로 "기억 클래스 + 자료형"이다.

분석 1, lv 1에서 ① declaration_specifiers에 대한 분석 결과

② type_specifier 단 하나 ("int")
② type_specifier ① declaration_specifiers
("int float" - 재귀적; 얼마든지 반복 가능)
② storage_class_specifier 단 하나 ("static")
② storage_class_specifier와 ① declaration_specifiers

네 가지의 경우가 있었고, 이는 재귀적이라 조합이 가능했다. 문법적으로는 맞을지언정 의미적으로 틀린 조합이 나올 수 있기에, 상식적인 c언어의 문법에서 의미적으로도 맞는 조합만 나열해보자면...

1. struct, union, enum, 또는 알려진 자료형(int, float, char, void) 단 하나
-> 우리는 흔히 "int a = 1;"과 같이 기억 클래스의 언급이 없이 사용하는데, 이 경우 컴파일러가 컴파일 타임에 기억 클래스를 결정한다.
(기억 클래스의 언급이 없을 경우 무조건 auto라 수업시간에도 교수님이 말씀하시기도 했다)

2. 기억 클래스 단 하나
-> "auto x;"와 같이 자료형의 언급이 없을 경우 컴파일러가 컴파일 타임에 자료형을 결정한다.
(typedef는 의미적으로 따져봤을 때 기억 클래스가 아니긴 하지만, 문법적으로 ② type_specifier의 위치가 맞긴 하기에 일단 넘어가겠다)

3. 기억 클래스 + 자료형
예를 들어 "static int a;"라 하면, 기억 클래스와 자료형을 모두 프로그래머가 지정해 주었기에, 컴파일러가 결정할 수 있는 것은 없다.

설명 2. declarator

딱 잘라서 한 문장으로 "이름" 이다.

분석 2, lv 3에서 ③ declarator에 대한 분석 결과
- 이름
- 이름인데 다른 이름을 가리킴
(가리키는 다른 이름의 자료형은 ① declaration_specifiers에서 나옴)

분석 2, lv 4에서 ④ direct_declarator에 대한 분석 결과
- 고유한 IDENTIFIER
- 고유한 배열의 이름
- 고유한 함수의 이름

이 두 분석을 합쳐보았을 때, ③ declarator는 결국
- 고유한 무언가 그 자체의 이름
- 고유한 무언가를 가리키는 것의 이름
이다.

결론

---

1. 포인터가 자료형보다는 이름에 더 가깝다는 점을 알게 되었다.

예를 들어서 정수 a를 가리키는 포인터 p가 있는데, 이를 c언어 코드로 나타낸다 해보자.

사람마다 코드정렬의 스타일이 다르겠지만, 전에 보았던 어떤 책의 스타일로는
"int* p = &a;"라 적을 수 있겠다.
필자가 현재 사용중인 ide(CLion)에서 이를 정렬하면
"int *p = &a;"로 만들어준다.

전자는 int를 가리키는 포인터임을 강조함에 의미가 있으나, c언어의 문법에 의하면 후자의 표현이 훨씬 알맞을 것이다. 포인터인지 아닌지는 자료형을 나타내는 declaration_specifiers가 아닌 이름을 나타내는 declarator에서 결정하기 때문이다. 왜 ide가 코드 정렬을 자꾸 이렇게 해주는지 그 이유를 드디어 알게 되었다.

2. 컴파일러의 경고를 더 잘 이해할 수 있게 되었다.

현대의 ide는 너무나 친절하기에 실시간으로 오류를 지적해주며 심지어 프로그래머 대신 수정을 해주기도 한다. 하지만 코딩테스트에서 사용하는 웹 페이지에는 이런 도움이 전혀 없다보니 오류 수정을 컴파일러의 경고에만 의존해야 한다. 실제로 코딩테스트를 푸는데 expression, unary, lvalue 등의 문법적인 용어들을 모르다보니, 컴파일러가 내는 오류를 이해하지 못한 상태로 버그를 잡는데 시간을 다 허비해서 문제를 제대로 풀지 못했다. 이제는 이런 경고를 이해할 수 있을 것 같다.